VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

USING INFINISPAN AS A BACKEND FOR CDI CONTEXTS

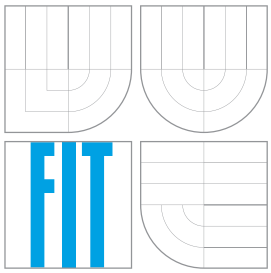
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

ADAM KÖVÁRI

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POUŽITÍ INFINISPANU PRO IMPLEMENTACI CDI KONTEXTU

USING INFINISPAN AS A BACKEND FOR CDI CONTEXTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ADAM KÖVÁRI
AUTHOR

VEDOUCÍ PRÁCE — RNDr. MAREK RYCHLÝ, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce vede čtenáře od vývoje kontextově závislé injekce v Java EE, ukazuje základní přehled o JSR 299, které se vyvinulo do rámce této specifikace a analyzuje referenční implementace Weld. V pozdějších částech je vytvořeno CDI rozšíření, které vytváří cluster-wide scope pomocí Infinispanu jako skladem dat, kterého základní vlastnosti jsou také prozkoumána.

Abstract

This thesis guides the reader from the evolution of the Contextual Dependency Injection in Java EE, shows the basic overview of JSR 299, frameworks that evolved into this specification and analyses the reference implementation Weld. In the later parts a CDI extension that creates a cluster-wide scope is creating using Infinispan as a backend storage, of which elementary features are also explored.

Klíčová slova

cdi, kontextuální, závislost, injekce, infinispan, nosql, cache, klaster

Keywords

cdi, contextual, dependency, injection, infinispan, nosql, cache, cluster

Citace

Adam Kövári: Using Infinispan as a Backend for CDI Contexts, bakalářská práce, Brno, FIT VUT v Brně, 2013

Using Infinispan as a Backend for CDI Contexts

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jozefa Hartingera

.....
Adam Kövári
May 9, 2013

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce, RNDr. Markovi Rychlému, Ph.D. stejně tak mému technickému vedoucímu Ing. Jozefovi Hartingerovi za veškerou podporu a pomoc v průběhu psaní této práce.

Contents

Chapter 1

Introduction

This theses starts with the history of Contextual dependency injecting and it's journey to the Enterprise specification world — chapter 2, then continues with the Java specification request(JSR) 299 overview — chapter 3 - where it tries to go through the history of this specification, the motivation to create it and relationships to other already existing standards and specifications. Later it explains basic concepts with focus on extensibility options which are later discussed in the implementation sections, beginning in JBoss Weld chapter 4 — the reference implementation of this specification and continuing in the implementation chapter which provides detailed overview of implementing a CDI extension in JBoss Weld environment.

Chapters describing the specification and basics of implementation of extensions will be further followed by description of Infinispan as a powerful data grid technology which is used as a back-end storage for clustered beans in this thesis — chapter 5. The CDI extension written as a part of this thesis will combine all these technologies in order to provide new `Context` for beans which wish to share their state across nodes in a cluster.

Last chapter 6 focuses on the extension itself, first the design ideas behind this exception — section 6.1, followed by the implementation part itself — section 6.2.

Chapter 2

Contextual Dependency Injection - the Journey to the Java EE

The history of contextual dependency injection as we know it today from the JSR 299 is a many years long and goes back to the early year 2004 when Martin Fowler[1] asked the readers of his site: when talking about Inversion of Control: „the question, is what aspect of control are they inverting?“. After talking about the term Inversion of Control Martin suggests renaming the pattern, or at least giving it a more self-explanatory name, and starts to use the term Dependency Injection. His article continues to explain some of the ideas behind Inversion of Control or Dependency Injection.

2.1 Spring Framework

Java EE back in the old days provided no such a thing as dependency injection or as previously often called – Inversion of Control(IoC) pattern. Java EE at the times of first Spring releases offered very complicated and un-intuitive support for transactional processes – EJBs 2.x. At this time a new framework called Spring was created in order to bring major simplification of writing business logic addressing both issues – EJBs and IoC pattern. This framework quickly became very popular and many Java EE developers were escaping to this framework.

2.2 javax.inject - JSR 330

Java EE community saw the success of the Spring's IoC pattern and tried to provide similar functionality in the EE environments. The first try was to copy the Spring's `@Autowired` functionality which in the EE specification is called `@Inject`.

2.3 Seam Framework

The Seam framework developed by the community around mostly JBoss projects such as Java Server Faces(JSF) greatly improved and simplified the development model around connecting the view-model of JSF and the business logic-model of EJBs. The Seam Frame-

[1] http://martinfowler.com/articles/injection.html

work offered many of functionality which later resulted in the JSR 299 specification and which later even replaced the Seam Framework as such.

2.4 CDI - JSR 299

JSR 299 represents the result of bringing the best of IoC patterns enriched by the contextual point of view into the Enterprise level specifications. The journey of this specification was quite interesting as shown above and is a great example of how different community projects and ideas can result into carefully prepared Enterprise-class specifications and standards that will exist and be supported as long as Java EE stays around.

Chapter 3

JSR 299 Overview

JSR 299 specification request provides interesting insight into topics such as why was this specification was created and which areas it aims to improve. Before this specification was created, integration of EJB backend model and JSF view model was mostly implemented in the Seam Framework[1] project. In order to standardize these integration services, JSR 299 was proposed with the following description:

JSR 299 being finalized from year 2007 to 2009 tries to enhance the EJB architecture that is defined as a component architecture for the development and deployment of component-based business applications. Particularly, EJB provides a programming model for components that make use of transactional resources. Applications written using this Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure.

This chapter is based on the overview chapter of the JSR-299: Contexts and Dependency Injection for the Java EE platform„[1].

The EJB component model still has some limitations, though:

- EJB components are unaware of the web-tier request, session and application contexts and have no access to state associated with those contexts. The lifecycle of a stateful EJB component may also not be scoped to a web-tier context.
- EJB components are not suitable for use in the presentation tier in general.
- JSF[2] provides no support for accessing transactional resources from managed bean components.
- The managed bean component model provides neither component-level nor method-level security.
- The context model provided by the servlet specification — and leveraged by JSF — is not rich enough for use in complex applications in an enterprise environment.

[1] Seam is a powerful open source development platform for building rich Internet applications in Java. Seam integrates technologies such as Asynchronous JavaScript and XML (AJAX), JavaServer Faces (JSF), Java Persistence (JPA), Enterprise Java Beans (EJB 3.0) and Business Process Management (BPM) into a unified full-stack solution, complete with sophisticated tooling.

[2] JavaServer Faces is a web-tier presentation framework which provides, among other facilities, a component model for graphical user interface components, a “managed bean„ component model for application logic, and an event-driven interaction model that binds the two component models. The managed bean component model is a contextual model where components are bound to one of the three web-tier contexts and may hold contextual state.

- The JSF component model is not consistent with Java EE component registry (JNDI), dependency injection, packaging and deployment standards.

JSR 299 specification enables EJB 3.0 components to be used as JSF managed beans by unifying the two component models and enabling a considerable simplification to the programming model for web-based applications in Java.

Particularly JSR 299 provides programming model suitable for rapid development of simple data-driven applications without sacrificing the full power of the Java EE 5 platform. This is a domain where Java EE has been perceived as overly complex.

Aspects considered in this specification include, but are not limited to, the following:

- Definition of additional capabilities to be used with the EJB component model, allowing EJB beans to act as JSF managed beans in a JavaServer Faces application. In principle, this is possible without requiring any changes to the EJB or JSF specifications.
- Definition of a unified annotations for manipulating contextual variables in a contextual, stateful, component-base architecture.
- Definition of an enhanced context model including business process contexts and conversational.
- Definition of an extension point that allows integration of business process management engines with the contextual component model.
- Integration of Java Persistence API extended persistence contexts with the enhanced context model.
- Ensure that components written conform to this specification may be executed in the context of a Web Services invocations.
- Ensure that the component model can be used with JSR-227 databindings.

3.1 Architecture

Following sections provide an overview of the specification in order to understand what programming model it brings. It is important to dive in into this development model so the value of the Clustering extension that this thesis brings can be seen. Sections below summarize the introductionary parts of the specification, Weld as the reference implementation and Seam as the Framework that inspired this specification.

The JSR 299 brings additional services which can help improve the structure of application code.

- Lifecycle definition for stateful objects bound to lifecycle contexts, where the set of contexts is extensible
- Typesafe, sophisticated dependency injection mechanism, having the ability to select dependencies at either development or deployment time, without verbose configuration

- The modularity of a Java EE application is taken into account when resolving dependencies between Java EE components
- Integration with the Unified Expression Language (EL), allowing any contextual object to be used directly within a JSP or JSF page
- Interceptors able to be associated to objects via typesafe interceptor bindings
- An event notification model
- The ability to decorate injected objects
- New web conversation context in addition to the three standard web contexts defined by the Java Servlets specification
- Clean integration with the container enabled through the possibility to write portable extensions using SPI

Services defined by this specification make it possible for objects to be bound to lifecycle contexts, to be injected and associated with decorators and interceptors, and to interact in a loosely coupled fashion by firing and observing events. Different kinds of objects are injectable, mainly EJB 3 session beans, managed beans and Java EE resources. General these objects are being referred to as beans and to instances of beans that belong to contexts as contextual instances. Contextual instances may be injected into other objects by the dependency injection service.

To make use of these services, additional bean-level metadata in the form of Java annotations and application-level metadata the form of an XML descriptor are available to the application developer.

Using these services significantly simplifies the task of creating Java EE applications by integrating the Java EE web tier with Java EE enterprise services. Particularly, EJB components may be used as JSF managed beans, therefore integrating the programming models of JSF and EJB.

There is even possibility to integrate with third-party frameworks. Portable extensions may provide objects to be injected or obtain contextual instances using the dependency injection service. The third-party framework may even raise and observe events using the event notification service.

An application that makes use of these services may be designed to execute in both the Java EE environment or the Java SE environment. If the application uses Java EE services like transaction management and persistence in the Java SE environment, the services are usually restricted to the subset defined for embedded usage by the EJB specification.

3.1.1 Contracts & responsibilities

The JSR 299 provides a very nice introduction into how the specification falls in the world of previous standards and specification. Below are the most important relationships described — passages of the specification.

The specification defines responsibilities of:

- the vendor implementing the functionality defined in this specification and providing a runtime environment in which the application executes

- the application developer using these services

The runtime environment is referred to as a container. An example of such a container would be a Java EE container or an embeddable EJB container.

3.1.2 Relationship to other specifications

Application developer can create container-managed components like EJBs, JavaBeans, servlets and then provide some additional metadata declaring additional behaviour that is defined in the JSR 299 specification. All those components may take advantage of the services this specification brings, combining the enterprise and presentational components defined by different Java EE technologies.

The specification additionally defines an SPI interface allowing alternative technologies to be integrated with the container and the Java EE environment, alternative web presentation technologies, for example.

Relationship to Java EE platform

In the Java EE 6 environment, all component classes that support injection, as defined by the Java EE 6 platform specification, can inject beans using the dependency injection service.

The Java EE platform specification defines a functionality for resource injection which exist in the Java EE component environment. Resources are identified by string-based names. This specification underlays that functionality by adding the possibility to inject an open-ended set of object types, including, but not limited to, component environment resources, based upon typesafe qualifiers.

Relationship to EJB

EJB provides a development model for application components which access transactional resources in a multi-user environment. EJB supports concerns such as role-based security, transaction demarcation, concurrency and scalability to be specified declaratively using annotations and XML deployment descriptors and enforced by the EJB container at runtime.

EJB components may be stateful, but are not contextual by nature. References to stateful component instances need to be explicitly passed between clients and stateful instances need to be explicitly destroyed by the application.

This specification improves the EJB component model with contextual lifecycle management.

Session beans instances obtained via the dependency injection service are contextual instances. They are bound to a lifecycle context and is available to other objects that execute in that context. The container automatically creates the instance when it is needed by a client. When the context ends, the container automatically destroys the instance.

Message-driven and entity beans are non-contextual objects by nature and may not be injected into other objects.

The container performs dependency injection on all session and message-driven bean instances, even those which are not contextual instances.

Relationship to managed beans

The Managed Beans specification provides the basic programming model for application components which are managed by the Java EE container.

As defined in this specification, most Java classes, including all JavaBeans, are managed beans.

This specification defines contextual lifecycle management and dependency injection as generic services which can be applied to all managed beans.

Any managed bean instance obtained using the dependency injection service is a contextual instance. It is bound to a lifecycle context and can be used by other objects which execute in that context. The container automatically creates the instance when it is needed by a client. When the context ends, the container automatically destroys the instance.

The container performs dependency injection on all managed bean instances, even those that are not contextual instances.

Relationship to Dependency Injection for Java

The Dependency Injection for Java specification provides a set of annotations for declaring injected fields, methods and constructors of beans. The dependency injection service uses these annotations.

Relationship to Java Interceptors

The Java Interceptors specification defines the basic programming model and semantics for interceptors. This specification improves this model by defining the ability to associate interceptors with beans using typesafe interceptor bindings.

Relationship to JSF

JavaServer Faces is a web presentation framework which provides a component model for the GUI components and an event-driven interaction model which binds user interface components to objects accessible through the Unified EL.

This specification allows any bean to be assigned a Unified EL name. Therefore, a JSF application can take advantage of the sophisticated context and dependency injection model provided by this specification.

3.2 Concepts

A Java EE component is a bean if the lifecycle of its instances may be managed by the container according to the lifecycle context model. A bean may carry metadata defining its lifecycle and interactions with other components.

In general, a bean is a source of contextual objects which define application state and/or logic. These objects are called contextual instances of the bean. The container creates and destroys these instances and assigns them to the appropriate context. Contextual instances of a bean may be injected into other objects (including other bean instances) that execute in the same context, and may be used in EL expressions that are evaluated in the same context.

Bean needs to comply to the following attributes:

- bean types, at least one

- qualifiers, at least one
- A scope
- Optionally, a bean EL name
- A set of interceptor bindings
- A bean implementation

More than that, a bean can but does not have to be an alternative.

Most of the time, a bean developer provides the bean implementation by writing business logic in Java code. The developer then needs to define the remaining attributes by explicitly annotating the bean class, or by allowing them to get default values by the container. In certain other cases-for example, Java EE component environment resources-the developer provides only the annotations and the bean implementation is provided by the container.

The bean types and qualifiers of a bean determine where its instances will be injected by the container, lookup and EL.

The bean developer may also create interceptors and/or decorators or reuse existing interceptors and/or decorators. The interceptor bindings of a bean determine which interceptors will be applied at runtime. The bean types and qualifiers of a bean determine which decorators will be applied at runtime.

3.2.1 Functionality provided by the container to the bean

A bean has the following capabilities provided by the container:

- transparent creation and destruction and providing a scope to a particular context,
- scoped resolution by name when used in a Unified EL expression,
- scoped resolution by bean type and qualifier annotation type when injected into a Java-based client,
- lifecycle callbacks and automatic injection of other bean instances, lookup and EL,
- method interception, callback interception, and decoration, and
- event notifications.

3.3 Portable extensions

Portable extensions may integrate themselves within the container by:

- Providing their own beans, interceptors and decorators to the container
- Injecting dependencies into their own objects using the dependency injection service
- Providing a context implementation for a custom scope
- Augmenting or overriding the annotation-based metadata with metadata defined in some other source

3.4 Packaging and deployment

When applications are started, the container must perform bean discovery, detect definition errors and deployment problems and raise events that allow portable extensions to be integrated with the deployment lifecycle.

During the bean discovery process, the following must be determined:

- The bean archives that exist in the application, and the beans they contain
- Which alternatives, interceptors and decorators are enabled for each bean archive
- The order of enabled interceptors and decorators

3.4.1 Bean archives

Deployed bean archives contain enabled bean classes.

- An EJB jar, a library jar, application client jar or rar[3] archive is a bean archive if it has a file named `beans.xml` in the `META-INF` directory.
- A directory in the JVM classpath is a bean archive if it has a file named `beans.xml` in the `META-INF` directory.
- The `WEB-INF/classes` directory of a war[4] is a bean archive if there is a file named `beans.xml` in the `WEB-INF` directory of the war.

It is not mandatory for a container to support application client jar bean archives.

A Java EE container is required by the Java EE specification to support Java EE modules. Other containers may decide whether they will provide support for war, EJB jar or rar bean archives.

The container tries to discover beans in all bean archives in the application classpath:

- In an application deployed as an ear[5], the container goes through every bean archive bundled with or referenced by the ear, including bean archives bundled with or referenced by wars and EJB jars contained in the ear. The bean archives could be library jars, EJB jars, rars or war `WEB-INF/classes` directories.
- In an application deployed as a war, the container goes through every bean archive bundled with or referenced by the war. The bean archives could be library jars or the `WEB-INF/classes` directory.
- In an application deployed as an EJB jar, the container goes through the EJB jar, if it is a bean archive, and every bean archive referenced by the EJB jar.
- An embeddable EJB container goes through each bean archive in the JVM classpath that is listed in the value of the embeddable container initialization property `javax.ejb.embeddable.modules`, or every bean archive in the JVM classpath if the property is not specified. The bean archives might be directories, library jars or EJB jars.

[3] Resource archive
[4] Web archive
[5] Enterprise archive

When discovering beans, the container considers:

- all Java classes in any bean archive,
- all `ejb-jar.xml` files in the metadata directory of any EJB bean archive,
- all Java classes referenced by the `@New` qualifier of an injection point of another bean,
- all interceptor or decorator classes declared in the `beans.xml` file of any bean archive.

When a bean class is deployed in two different bean archives, non-portable behaviour results. Portable applications must deploy each bean class in one bean archive at most.

Additional beans can be registered programmatically with the container by the application or a portable extension after the automatic bean discovery completes. Portable extensions can even integrate with the process of building the `Bean` object for a bean, to enhance the container's built-in functionality.

Chapter 4

JBoss Weld – reference implementation of JSR 299

Weld is the reference implementation (RI) for JSR-299: *Java Contexts and Dependency Injection for the Java EE platform (CDI)*. CDI is the Java standard for dependency injection and contextual lifecycle management, led by Gavin King for Red Hat, Inc. and is a Java Community Process (JCP) specification that integrates cleanly with the Java EE platform. All Java EE 6 application server provide support for JSR-299 (including the web profile).

The Weld – JSR-299 Reference Implementation[2] provides great introduction in the JBoss Weld as the Reference Implementation of the JSR-299 and this chapter is greatly inspired by this guide.

4.1 Seam 2 Framework

Seam is an application framework for Java EE. As discussed earlier Seam played an irreplaceable role in designing the CDI standard as many parts of Seam directly resulted in the CDI specification. Therefore understanding the basic principles the Seam is built upon is crucial in order to understand the core of CDI functionality and functionality in the enterprise applications for Java EE 6.

Please note that this chapter is inspired by publication "A Framework for Enterprise Java„[3].

The Seam Framework is built up on the following principles:

- One kind of "stuff„

  Seam provides a uniform component model for all business logic in an application. A Seam component can be stateful, with the state associated with any one of several well-defined contexts, such as the long-running, persistent, business process context and the conversation context, which is preserved across multiple web requests in a user interaction.

  Presentation tier components and business logic are not distinct components in Seam. An application can be layered according to whatever architecture you prefer, rather than being forced to make your application logic compliant with an unnatural layering scheme forced upon you by whatever combination of any frameworks used today.

Unlike plain Java EE or Java EE components, Seam components may simultaneously access state associated with the web request and state held in transactional resources (without the need to propagate web request state manually via method parameters). There might be objections that the application layering imposed upon you by the old Java EE platform was a Good Thing. However, nothing stops the developer from creating an equivalent layered architecture using Seam, the difference is that the developer gets to architect his/her own application and decide what the layers are and how they work together.

- Integrate JSF with EJB 3.0

  JSF and EJB 3 are two main new features of Java EE 5. EJB 3 is a brand new component model for server side business and persistence logic. JSF is a great component model for the presentation tier. Unfortunately, neither of these two component models is able to solve all problems in development of business applications by itself. It is true that JSF and EJB3 work best used together. But the Java EE 5 specification provides no standard way to integrate the two component models. Fortunately, the creators of both models foresaw this situation and provided standard extension points to allow extension and integration with other frameworks.

  Seam unifies the component models of JSF and EJB 3, eliminating glue code, and letting the developer think about the actual business logic.

  It is possible to write Seam applications where "everything„ is an EJB. This may seem as a surprise if a developer is used to thinking of EJBs as coarse-grained, so-called "heavyweight„ objects. However, version 3.0 has completely changed the nature of EJB from the point of view of the developer. An EJB is a fine-grained object, nothing more complex than an annotated JavaBean. Seam even encourages you to use session beans as JSF action listeners!

  On the other hand, if the developer prefers not to adopt EJB 3.0 at this time, he/she doesn't have to. Virtually any Java class may be a Seam component, and Seam provides all the functionality that is expected from a "lightweight„ container, and more, for any component, EJB or otherwise.

- Integrated with Java EE6

  While Seam 2.2 target Java EE 5 mainly, some of Java EE 6 technologies are available on Seam 2.3.x.

  Seam 2 and some of its extensions/implementations were added into Java EE 6 as CDI technology. So this should be a current focus of majority users. But for previous Seam 2.2 users who don't want or can't use pure Java EE 6, the Seam framework brings some new features from the Java EE 6, for instance, JSF 2, JPA 2 and Bean Validation integrations into Seam 2.3.x.

- Integrated AJAX

  Seam supports some of the best open source JSF-based AJAX solutions: RichFaces and ICEfaces. These solutions let the developer add AJAX capability to the user interface without the need to write a line of JavaScript code.

  Alternatively, Seam provides a built-in JavaScript remoting layer that lets the developer call components asynchronously from client-side JavaScript without the need for

an intermediate action layer. The application can even subscribe to server-side JMS topics and receive messages via AJAX push.

Neither of these approaches would work well, if there was no Seam's built-in concurrency and state management, which ensures that many concurrent fine-grained, asynchronous AJAX requests are handled safely and efficiently on the server side.

- Business process as a first class construct

  Optionally, Seam provides transparent business process management via jBPM. It is very easy to implement complex workflows, collaboration and task management using jBPM and Seam.

  Seam even allows the developer to define presentation tier pageflow using the same language (jPDL) that jBPM uses for business process definition.

  JSF provides an incredibly rich event model for the presentation tier. Seam enhances this model by exposing jBPM's business process related events via exactly the same event handling mechanism, providing a uniform event model for Seam's uniform component model.

- Declarative state management

  The developers are used to the concept of declarative transaction management and declarative security from the early days of EJB. EJB 3.0 even introduces declarative persistence context management. These are three examples of a broader problem of managing state that is associated with a particular context, while ensuring that all needed cleaning up actions occurs when the context ends. Seam takes the concept of declarative state management much further and applies it to application state. Traditionally, Java EE applications implement state management manually, by getting and setting servlet session and request attributes. This approach to state management is the source of many bugs and memory leaks when applications fail to clean up session attributes properly, or when session data associated with different workflows collides in a multi-window application. Seam has the potential to almost entirely eliminate this class of bugs.

  Declarative application state management is made possible by the richness of the context model defined by Seam. Seam extends the context model defined by the servlet specification - request, session, application – with two new contexts – conversation and business process — that are more meaningful from the point of view of the business logic.

  The developer will notice how many things become easier once he/she starts using conversations. It has been common to suffer a pain from dealing with lazy association fetching in an ORM solution like Hibernate or JPA. Seam's conversation-scoped persistence contexts mean that a developer rarely gets to see a LazyInitializationException. The typical problems, such as problems with the refresh button, the back button, the duplicate form submission, the propagating message across a post-then-redirect are solved using the Seam's conversation management without the developer even needing to really think about them. They're all symptoms of the broken state management architecture that has been prevalent since the earliest days of the web.

- Bijection

The notion of Inversion of Control or dependency injection exists in both JSF and EJB3, as well as in numerous so-called "lightweight containers„. Most of these containers emphasize injection of components that implement stateless services. Even when injection of stateful components is supported (such as in JSF), it is virtually useless for handling application state because the scope of the stateful component cannot be defined with sufficient flexibility, and because components belonging to wider scopes may not be injected into components belonging to narrower scopes.

Bijection differs from IoC in that it is dynamic, contextual, and bidirectional. The developer can think of it as a mechanism for aliasing contextual variables (names in the various contexts bound to the current thread) to attributes of the component. Bijection allows auto-assembly of stateful components by the container. It even allows a component to safely and easily manipulate the value of a context variable, just by assigning it to an attribute of the component.

- Workspace management and multi-window browsing

  Seam applications let the user freely switch between multiple browser tabs, each associated with a different, safely isolated, conversation. Applications may even take advantage of workspace management, allowing the user to switch between conversations (workspaces) in a single browser tab. Seam provides not only correct multi-window operation, but also multi-window-like operation in a single window!

- Prefer annotations to XML

  Traditionally, the Java community has been in a state of deep confusion about precisely what kinds of meta-information counts as configuration. Java EE and popular "lightweight„ containers have provided XML-based deployment descriptors both for things which are truly configurable between different deployments of the system, and for any other kinds or declaration which can not easily be expressed in Java. Java 5 annotations changed all this.

  EJB 3.0 embraces annotations and "configuration by exception„ as the easiest way to provide information to the container in a declarative form. Unfortunately, JSF is still heavily dependent on verbose XML configuration files. Seam extends the annotations provided by EJB 3.0 with a set of annotations for declarative state management and declarative context demarcation. This lets the developer to eliminate the noisy JSF managed bean declarations and reduce the required XML to just that information which truly belongs in XML (the JSF navigation rules).

- Integration testing is easy

  Seam components, being plain Java classes, are by nature unit testable. But for complex applications, unit testing alone is insufficient. Integration testing has traditionally been a messy and difficult task for Java web applications. Therefore, Seam provides for testability of Seam applications as a core feature of the framework. The developer can easily write JUnit or TestNG tests that reproduce a whole interaction with a user, exercising all components of the system apart from the view. The developer can run these tests directly inside an IDE, where Seam will automatically deploy EJB components using the Arquillian testing framework.

- The specifications are not perfect

The Seam development team thinks the latest incarnation of Java EE is great. But it is also clear that it's never going to be perfect. Where there are holes in the specifications (for example, limitations in the JSF lifecycle for GET requests), Seam fixes them. And the authors of Seam are working with the JCP expert groups to make sure those fixes make their way back into the next revision of the standards.

- There's more to a web application than serving HTML pages

  Today's web frameworks think too small. They let the developer to get user input off a form and into your Java objects. And then they leave you hanging. A truly complete web application framework should address problems like persistence, concurrency, asynchronicity, state management, security, email, messaging, PDF and chart generation, workflow, wikitext rendering, webservices, caching and more. Once the developer scratches the surface of Seam, many problems become simpler...

  Seam integrates JPA and Hibernate for persistence, the EJB Timer Service and Quartz for lightweight asychronicity, jBPM for workflow, JBoss Rules for business rules, Meldware Mail for email, Hibernate Search and Lucene for full text search, JMS for messaging and JBoss Cache for page fragment caching. Seam layers an innovative rule-based security framework over JAAS and JBoss Rules. There's even JSF tag libraries for rendering PDF, outgoing email, charts and wikitext. Seam components may be called synchronously as a Web Service, asynchronously from client-side JavaScript or Google Web Toolkit or, of course, directly from JSF.

4.2 CDI, Weld and Seam – How do they relate to each other?

It may seem confusing how do these "terms„(CDI, Weld and Seam) relate to each other. Here is a short summary about differences and things in common between them:

- CDI is a JCP specification included in Java EE
- Weld is the reference implementation of CDI
- Seam 3 is a set of modules which extend CDI to provide functionality beyond that offered by Java EE 6

The two technologies share a common goal of providing a unified, contextual, programming model for Java Web Applications. Both provide integration of EJB and JSF.

However, Seam 3 is a superset of JSR-299. JSR-299 can be thought of as the core of Seam 3 – it's the basic programming model for your application components, and for the built-in components that make up the Seam framework. Weld is our implementation of this programming model. Based on this programming model, Seam provides a full framework for application development, including integration with various non-standard open source technologies. JSR-299 defines a very powerful framework for portable extensions. Seam 3 is implemented as a set of portable extensions, or modules, for JSR-299, that run in any environment which supports JSR-299 (including any Java EE 6 environment).

The following diagram illustrates the relationship between Seam 2, Seam 3 and JSR-299 (implemented by Weld):

Seam will continue to be the vehicle which delivers BPM integration, Seam Security, Drools integration, RESTeasy integration, PDF and email templates, Excel generation, etc. The Seam can be thought of as "the goodies„. Seam will also contain improvements for technologies in Java EE such as JSF fixes or Seam-managed persistence.

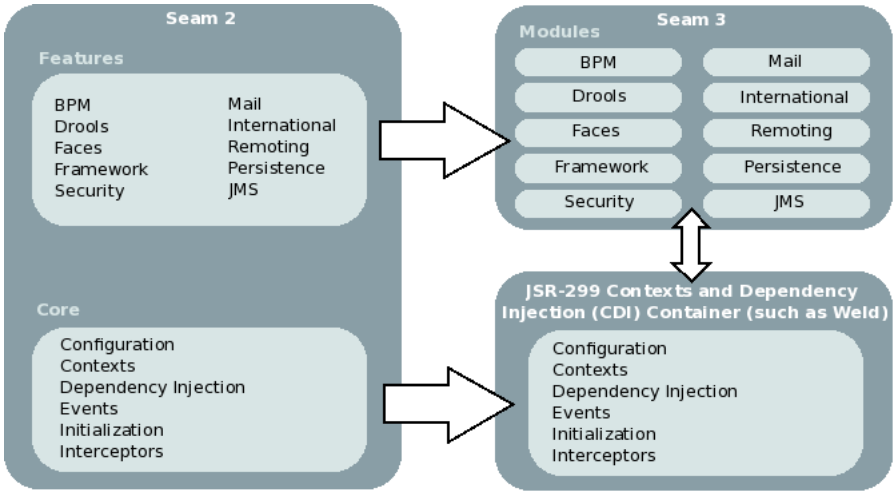


Figure 4.1: CDI is the specification, of which Weld is a full implementation.

4.3 Deployment

Weld supports these platforms at the time of writing these thesis:

- JBoss Application Server 7
- JBoss Application Server 6
- GlassFish V3.x
- Apache Tomcat 6 & 7
- Jetty 6.1 & 7.x
- Java SE 5.0+

4.4 Extensibility

Apache DeltaSpike consist of a number of portable CDI extensions that provide useful features for Java application developers.

The DeltaSpike project will also ensure true portability. DeltaSpike is tested on different CDI implementations like Apache OpenWebBeans and JBoss Weld, and also on different Java EE servers like Apache Tomcat and TomEE, JBoss-AS7, Oracle GlassFish 3.1+, IBM Websphere 8.x, Oracle Weblogic Server 12c, Jetty, and others.

Chapter 5

Infinispan – Transactional in-memory key/value NoSQL datastore & Data Grid

Infinispan[4] is very scalable, highly available key/value NoSQL datastore and distributed data grid platform - 100% open source, and written in Java. The purpose of Infinispan is to provide a data structure that is highly concurrent, designed ground-up to make use of the most of modern multi-processor/multi-core architectures while at the same time providing distributed cache capabilities.

Infinispan User Guide[4] is a resourceful source of knowledge about Infinispan as such and also shares multiple deep-level details behind the implementation and design choices that make Infinispan so great. This chapter leverages this publication greatly.

At its core level, Infinispan provides a Cache interface which extends `java.util.Map`. It is also optionally backed by a peer-to-peer network architecture to distribute state efficiently around the data grid.

Infinispan offers high availability through replicating state across a network as well as optionally persisting state to configurable cache stores, Infinispan offers enterprise features such as efficient eviction algorithms to control memory usage as well as JTA compatibility — to support transactional semantics.

In addition to the peer-to-peer architecture of Infinispan, a client/server mode is also supported. This provides the ability to run farms of Infinispan instances as servers and connecting to them using a plenty of clients — both written in Java as well as other popular open source and proprietary platforms.

5.1 Introduction

What makes Infinispan so interesting:

- State-of-the-art core - Infinispan's core is a specialised data structure, tuned to and geared for a great degree of concurrency — especially on multi-CPU/multi-core architectures. Most of the internals are essentially lock- and synchronization-free, favouring state-of-the-art non-blocking algorithms and techniques wherever possible. Even though non-clustered caching (LOCAL mode) is not its primary goal, Infinispan still is very competitive, as demonstrated in the benchmarks.

- Massive heap - If you have 100 blade servers, and each node has 2GB of space to dedicate to a replicated cache, you end up with 2 GB of total data. Every server is just a copy. On the other hand, with a distributed grid — assuming the user wants 1 copy per data item — he/she gets a 100 GB memory backed virtual heap that is efficiently accessible from anywhere in the grid. Session affinity is not required, so there is no need for fancy load balancing policies. Of course they can still be used for further optimisation. If a server fails, the grid simply creates new copies of the lost data, and puts them on other servers. This means that applications looking for ultimate performance are no longer forced to delegate the majority of their data lookups to a large single database server — that massive bottleneck that exists in over 80% of enterprise applications!

- Extreme scalability - Since data is evenly distributed, there is essentially no major limit to the size of the grid, except group communication on the network – which is minimised to just discovery of new nodes. All data access patterns use peer-to-peer communication where nodes directly speak to each other, which scales very well.

- Not Just for Java (PHP, Python, Ruby, C, etc.) - Infinispan ships with a language-independent server module. This supports the popular memcached protocol — with existing clients for almost every popular programming language — as well as an optimised Infinispan-specific protocol, called Hot Rod. And finally, Infinispan also has a REST API. This means that Infinispan is not just useful to Java or JVM-based applications. Any major website or application that wants to take advantage of a fast data grid will be able to do so.

- Support for Compute Grids - Also on the roadmap is the ability to pass a Runnable around the grid. The developer will be able to push complex processing towards the server where data is local, and pull back results using a Future. This map/reduce style paradigm is common in applications where a large amount of data is needed to compute relatively small results.

- Management is the key! - When the developer starts thinking about running a grid on several hundred servers, management is no longer an extra, it becomes a necessity. This is on Infinispan's roadmap. The Inifinispan team aims to provide rich tooling in this area, with many integration opportunities.

5.2 Relationship to CDI and Clustered context

Infinispan includes integration with CDI in the infinispan-cdi module, as shown in listing 5.1. Configuration and injection of the Inifispan's Cache API is provided, and it is planned to bridge Cache listeners to the CDI event system. The module also provide partial support of the JCache (JSR-107) caching annotations.

```
...
import javax.inject.Inject;

public class GreetingService {
    @Inject
    private Cache<String, String> cache;

    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

Figure 5.1: A simple example of injecting an embedded cache

As can be seen, in listing 5.1, Infinispan already provides support for CDI – at least for injecting Caches. This support is not used in the Extension this theses is about. The reason is that the CDI support provided by the Infinispan is yet to evolve, it doesn't provide enough configuration options yet.

Chapter 6

Implementation

6.1 Using Infinispan as a Backend for CDI Contexts – design overview

At this point it should be clear what CDI is and how it works. Advantages of Infinispan as a Key-Value storage have been mentioned as well. In combination of these 2 technologies it is possible to achieve creating a powerful, yet not too complicated, extension that brings a new `Context`, the one that allows sharing the state of a bean throughout the whole Java Application Server Cluster.

The basic idea is simple, you have a bean and want it's state to be seamlessly shared across all the nodes, without having to worry about manually synchronizing the state. The user just needs to use some annotations to add the needed metadata, such as which bean should be clustered this way and which operations affect the shared state.

The next chapter will explain in the depth the details of creating this extension. It mentions the issues that needed to be overcome and lessons learned during the implementation phase as well.

6.2 Implementation overview

Currently Java EE CDI specification JSR-299 or any other does not define a scope that would allow to share a state of beans across an application server's cluster nodes.

Infinispan provides a clustered key-value storage database which can be used to hold this state.

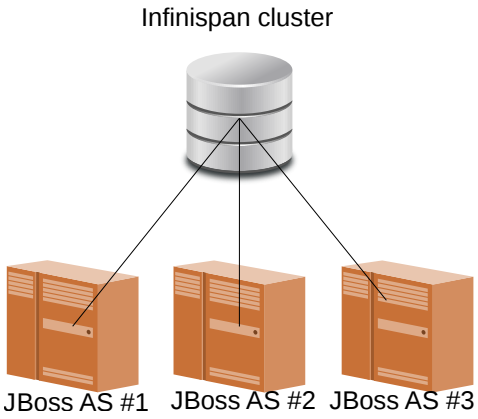


Figure 6.1: Typical architecture with `@ClusterScoped`

Implementation of `@ClusterScoped` will consist of implementing/extending following:

- CDI extension itself - `Context`, `Scope`, `Extension`
- Configuration options of the extension - via a `.properties` files
  - Default replicated cache - needs to be created using the JBoss CLI and configured appropriately
- Concurrent access to the instance of a clustered bean - CDI `@Interceptors`
- Definition of a lifecycle for a clustered bean - bound to Infinispan's cache lifecycle
  - When is the bean instantizied - with first injection on any node
  - When is the bean destroyed - with destruction of the cache — either persistent or not persistent

6.3 Creating the extension

Creating a portable extension starts with writing a class implementing the `Extension` interface, see listing 6.2. This interface does not define any methods but i's required by the Java SE service provider architecture.

```
class MyExtension implements Extension {
...
}
```

Figure 6.2: Extension class

Next thing that needs to be done is to register the extension as a service provider by creating a file named `META-INF/services/javax.enterprise.inject.spi.Extension`, which contains the name of our extension class, see listing 6.3.

```
org.example.extension.MyExtension
```

Figure 6.3: Extension service file

An extension itself is not a bean, exactly, as it is instantiated by the container during the initialization process which takes place before any beans or contexts exist. However, it can be injected, see listing 6.4, into other beans when the initialization process is complete.

```
@Inject
public MyBean(MyExtension myExtension) {
   myExtension.doSomething();

}
```

Figure 6.4: Injection of `Extension` into other bean

And, like beans, extensions can have observer methods. Usually, the observer methods observe container lifecycle events.

6.3.1 Container lifecycle events

The container fires a series of events during the initialization process, including:

- `BeforeBeanDiscovery`
- `ProcessModule`
- `ProcessAnnotatedType` and `ProcessSyntheticAnnotatedType`
- `ProcessInjectionTarget` and `ProcessProducer`
- `ProcessInjectionPoint`
- `ProcessBeanAttributes`
- `ProcessBean`, `ProcessManagedBean`, `ProcessSessionBean`, `ProcessProducerMethod` and `ProcessProducerField`
- `ProcessObserverMethod`
- `AfterBeanDiscovery`
- `AfterDeploymentValidation`

Extension may be observer using the `@Observes` annotation, as shown in the listing 6.5.

```
class MyExtension implements Extension {
   void beforeBeanDiscovery(@Observes BeforeBeanDiscovery event) {
      log.debug("the scanning process begins");
   }

   <T> void processAnnotatedType(@Observes ProcessAnnotatedType<T> event) {
      log.debug("type scanned: " + event.getAnnotatedType()
        .getJavaClass().getName());
   }

   void afterBeanDiscovery(@Observes AfterBeanDiscovery event) {
      log.debug("the scanning process completed");
   }
}
```

Figure 6.5: Extension observing events using the observer methods

Extensions can do much more than just observe events. The extension is allowed to modify the container's metamodel and more. A very simple example can be found in the listing 6.6.

```
class MyExtension implements Extension {
   <T> void processAnnotatedType(@Observes ProcessAnnotatedType<T> event) {
      //tell the container to ignore the type if it is annotated @Ignore
      if ( event.getAnnotatedType().isAnnotionPresent(Ignore.class) ) {
        event.veto();
      }
   }
}
```

Figure 6.6: Extension observing the container's lifecycle events

The observer method may inject a `BeanManager` as shown in the listing 6.7.

```
<T> void processAnnotatedType(@Observes ProcessAnnotatedType<T> event,
    BeanManager beanManager) {
}
```

Figure 6.7: Injecting the `BeanManager` into an observer method

Extension observer methods are not allowed to inject any other object.

6.4 Technologies used

Implementation of `@ClusterScoped` extensions makes use of following technologies:

- Maven - as a build tool
- JBoss Weld Solder - set of classes/tools simplified development of CDI extensions
- Apache DeltaSpike - set of CDI extensions and tools simplifying the development of CDI extensions

- Infinispan - key/value storage used to store the state of beans annotated as `@ClusterScoped`
- Java EE 6 compatible application server — tested in the JBoss Enterprise Application Platform 6
- Arquillian - testing framework

6.5 Structure of the extension

Figures 6.8 and 6.9 show the structure of the source codes of the extension.

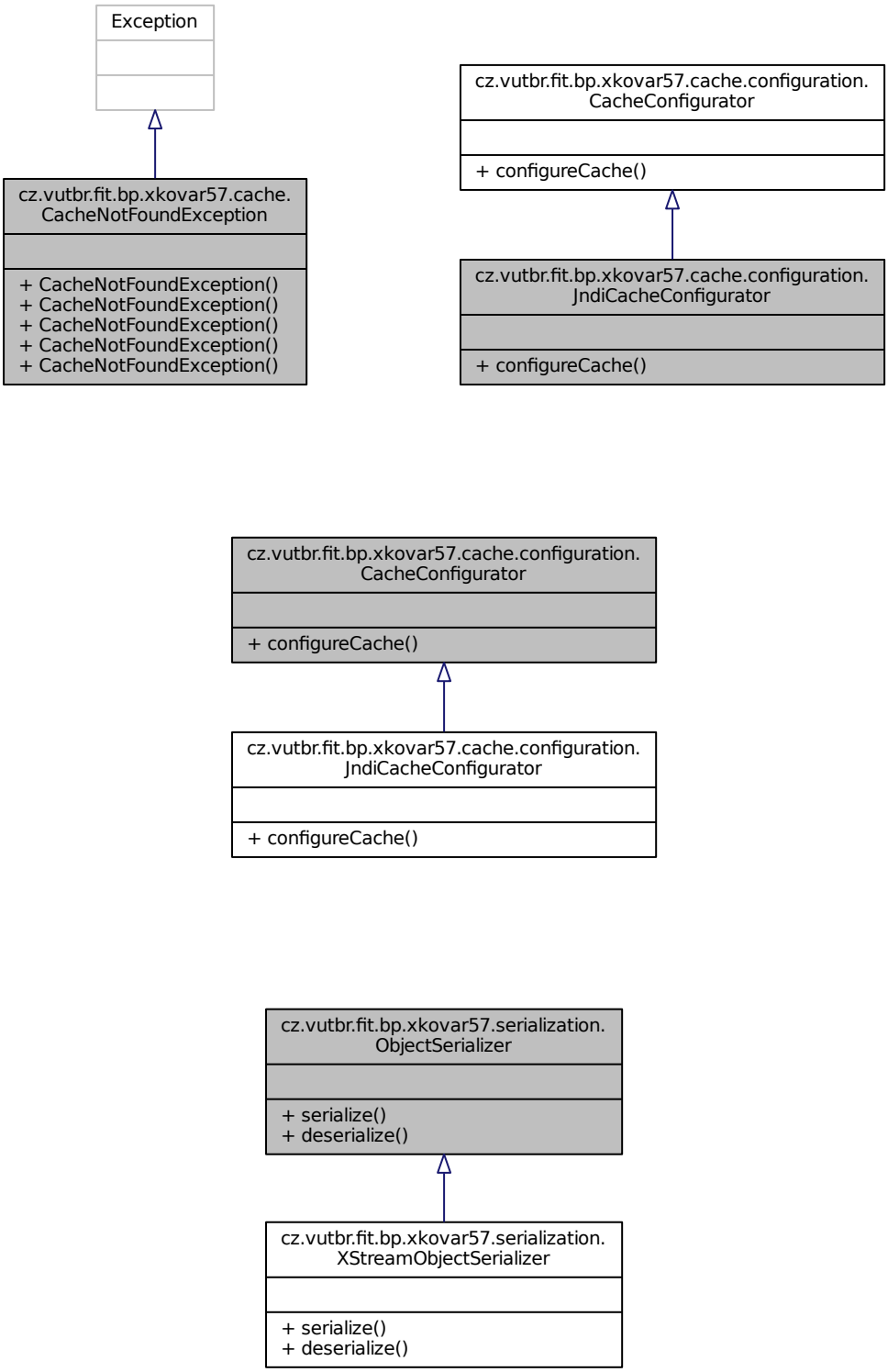


Figure 6.8: Class diagrams of the extension classes #1

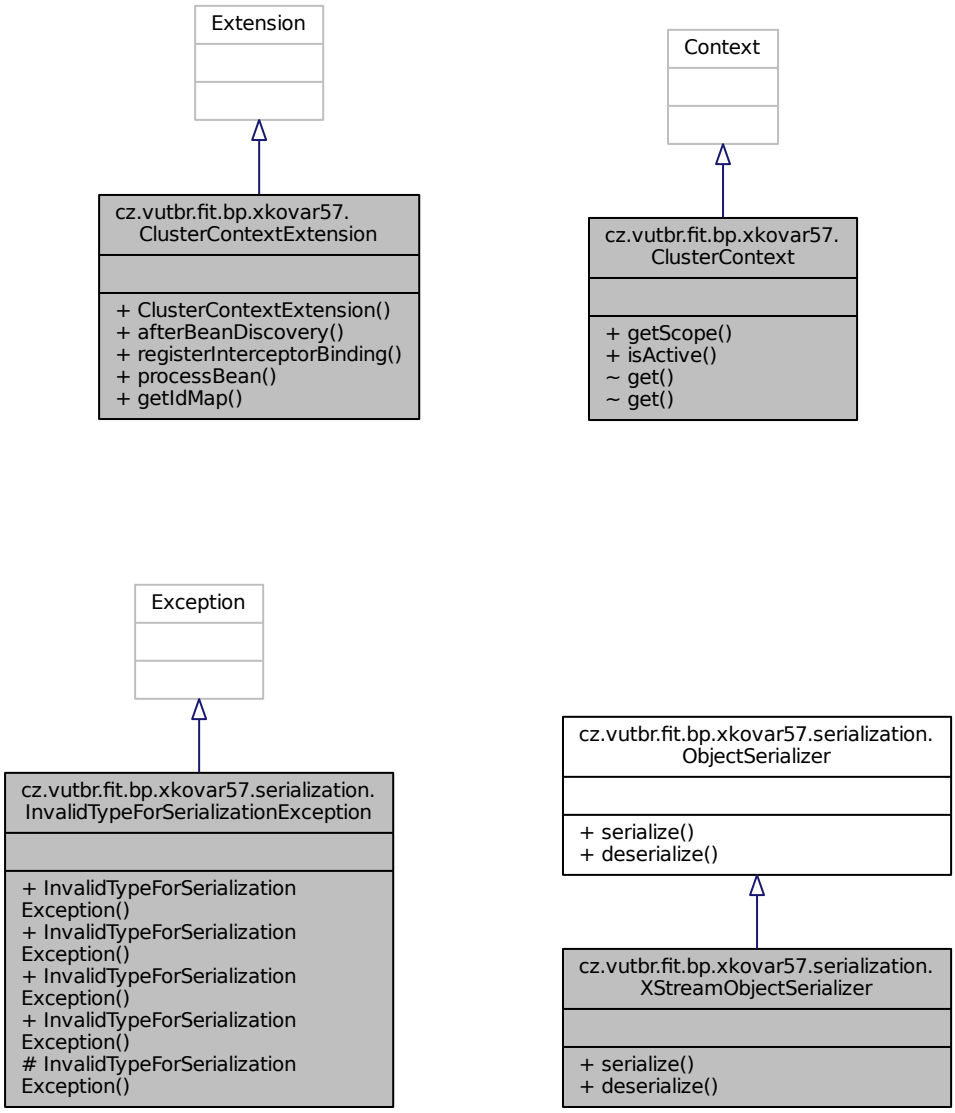


Figure 6.9: Class diagrams of the extension classes #2

6.6 Source codes

The source codes of all the extension itself, the application and this thesis are publicly available from public GIT repositories:

- extension source code:
  https://bitbucket.org/kovariadam/cluster-cdi

- testing application:
  https://bitbucket.org/kovariadam/cluster-cdi-test
- this thesis:
  https://bitbucket.org/kovariadam/cdi-cluster-doc

6.7 Code in depth

The source code is available under the `cz.vutbr.fit.bp.xkovar57` package. The code is structured further in packages as shown in the appendix A.

6.7.1 Object serialization

The object serialization has caused the most issues in this extension. First problem is in-ability to change the ClassLoader in the JBoss modular classloader architecture to use the contextual classloader of the application to serialize/deserialize objects saved into the Infinispan cache which resulted in the need to perform the serialization/deserialization programatically in the extension itself. I have created a community thread in order to try to answer this question in the JBoss forum [1].

This was already a workaround of the original classloader issue, however even this was not simple as the Infinispan is unable to handle comparision of arrays of primitive types, since it always uses `.equals()` method to compare values in the cache, regardless whether the type of the object is primitive array or not. I have reported this bug/enhancement request against the Infinispan project in the ISPN-3050[2] ticket.

6.7.2 `@ClusterScoped` Interceptor

This interceptor is invoked whenever a method annotated as `@ClusteredOperation` is invoked. The reason for intercepting these methods' invocations it to make sure that the invocation happens on the cluster-wide copy of the bean and the changes are properly reflected in other nodes, once they will access a method annotated with the `@ClusteredOperation` as well.

The logic is pretty simple, the clustered copy of the bean is downloaded from the Infinispan cluster, the operation is performed on this copy and if in the meantime the clustered copy got changed from another node, the operation is repeated until the local copy can safely be again stored to the Infinispan cluster. This all happens in a transactional-fashion to make sure the data is consistent all the time.

The tricky part of this interceptor is that Infinispan when serializing a bean, it also serializes annotations the bean has, which means that during the deserialization this interceptor would get invoked recursively for infinite amount of time — well, until a `StackOverflown` exception gets thrown from the JRE. In order to prevent this recursion, the interceptor maintains a `ThreadLocal` variable that holds the stack of the invocations on this bean and if the invocation is repeated, then the interceptor is not processed anymore, the flow is returned to the next interceptor of the interceptor chain.

[1] https://community.jboss.org/thread/224280
[2] https://issues.jboss.org/browse/ISPN-3050

6.7.3 Extension's beans descriptor `META-INF/beans.xml`

The extension's beans descriptor defines implementations of `CacheConfigurator` interface and the `ObjectSerializer`. This must be configured appropriately for the given deployment inside the extension's JAR file. The default version of the `META-INF/beans.xml` descriptor is shown in the listing 6.10.

```
<beans>
    <alternatives>
        <class>cz.vutbr.fit.bp.xkovar57.cache.configuration.
            JndiCacheConfigurator</class>
        <class>cz.vutbr.fit.bp.xkovar57.serialization.XStreamObjectSerializer</
            class>
    </alternatives>
</beans>
```

Figure 6.10: Extensions bean descriptor `META-INF/beans.xml`

6.8 Example deployment

The simplest web deployment(WAR) of the extension could have a structure as shown in the appendix B.

6.8.1 Most important parts

Use of the `@ClusterScoped` annotation

`ClusteredBean.java`

```
@ClusterScoped
public class ClusteredBean implements Serializable {
...
}
```

Figure 6.11: Example clustered bean

The bean shown in the listing 6.11 bean will be stored in a cache and support the `@ClusteredOperations`.

Use of the `@ClusteredOperation` annotation

`ClusteredBean.java`

```
@ClusteredOperation
public String getValue() {
    return value;
}
```

Figure 6.12: Example method of a clustered bean that supports clustered operations

When the method in the listing 6.12 gets invoked, the state is loaded and saved from and to the clustered cache.

Use of the `@Inject` of the bean

`GetServlet.java` / `SetServlet.java`

```
@Inject
private ClusteredBean clusteredBean;
```

Figure 6.13: Injection of the clustered bean

The clustered bean can be injected as any other CDI bean, see listing 6.13.

Configuration of JNDI cache

`META-INF/cdi-cache-jndi.properties`

```
jndi.name=java:jboss/cache/container/cdi
cache.name=
```

Figure 6.14: JNDI cache configuration properties file

In the listing 6.14 the JNDI name of the cache and it's name can be configured. Empty value means default cache.

Configuration of interceptors

`beans.xml`

```
<interceptors>
    <class>cz.vutbr.fit.bp.xkovar57.interceptors.ClusteredOperationInterceptor</
        class>
</interceptors>
```

Figure 6.15: Application's `beans.xml` descriptor with the `ClusteredOperationInterceptor` enabled

The application's beans.xml shown in the listing 6.15 shows enabled `ClusteredOperationInterceptor` interceptor for the deployment.

6.8.2 Extension configuration

The extension needs following configuration in order to work:

- enabling the `ClusteredOperationInterceptor` the application's `beans.xml`, listing 6.15
- the JNDI cache configuration, if JNDI provider is used `META-INF/cdi-cache-jndi.properties`, listing 6.14

- creating a cache, listing 6.16
- the alternatives in the extension's `beans.xml`, subsection 6.7.3

6.8.3 Creating a replicated cache with the JBoss CLI

To create a replicated Infinispan cache, one can use the the JBoss CLI command shown in listing 6.16.

```
/subsystem=infinispan/cache-container=web:write-attribute(name=jndi-name, value=
    cache/container/web)
/subsystem=infinispan/cache-container=web/replicated-cache=cdi-cluster:write-
    attribute(name=start, value=EAGER)
```

Figure 6.16: JBoss CLI commands to create a replicated Cache

6.8.4 Trace logging

To enable the trace logging of the extension, one can use the JBoss CLI command shown in listing 6.17.

```
/subsystem=logging/logger=cz.vutbr.fit.bp.xkovar57:add(level=ALL)
```

Figure 6.17: JBoss CLI command to enable the TRACE logging for the extension

6.8.5 Implementation of the `@ClusteredOperations`

`CacheProvider.java`

The pseudo-code responsible for invoking clustered operations on a clustered bean is shown in the listing 6.18.

```
boolean success = false;
int maxTries = 5;
int i = 0;
Object returnValue = null;

while (success != true && i < maxTries) {
    Object currentlyCachedVersion = cache.get(key);
    returnValue = method.invoke(target, args);

    success = cache.replace(key, currentlyCachedVersion, objectSerializer.
        serialize(target));
    i++;
}

if (i == maxTries && !success) {
    logger.warning("Failed to update " + key);
}
```

Figure 6.18: Invoking a clustered operation pseudocode

This code first retrieves the current version of the object from the cache, it invokes the method on it that has been intercepted due to being annotated as `@ClusteredOperation` and then it saves it in the cache. If the `replace` operation fails, that means that the cache has been update between the old object was retrieved and now that it is being saved. Then the whole operation repeats – the old version is loaded and the method is again invoked on the latest version of the bean. This prevents any inconsistency that could occur. There is a limitation of how many times this operation can be repeated though – this is in place in order to prevent any deadlock situations.

6.9 Testing

Automated testing of every application is crucial nowadays. The extension comes with several options of verifying it's proper functionality.

6.9.1 Manual testing

The extension comes with a very a simple testing application which is described in the section 6.8. This application consists of two servlets, one of them updates the cache, the other one reads the current value. They use `java.util.Calendar` instance as a value that gets printed on a server log whenever any of these servlets gets called.

Once the application is deployed, it can be tested as:

- `http://<serverURL>:<serverport>:8180/cdi-cluster-test/get`
- `http://<serverURL>:<serverport>/cdi-cluster-test/set`

Where the `<server URL>:<server port>` needs to be replaced for the actual values. The `set` servlet updates the value of the clustered bean, while the `get` prints out the current value.

6.9.2 Unit and integration testing

The extension also contains a very basic skeleton for an Arquillian test case, however, this is nowhere nearly finished as the Arquillian does not proved any documentation related to deploying Infinispan caches in the managed JBoss AS environment. This area is a good opportunity for future improvements though. It would be interesting to perform some performance benchmark of this extension.

6.10 Summary

The sections above provide step-by-step guide on how this extension was implemented. First, it was important to understand how does container communicate with extensions through it's lifecycle events, as shown in section 6.3.1. Later, most of every new software today uses different libraries and components, these technology choices were shown in section 6.4.

Following these sections, the thesis gets deeper in the code itself, starting in section 6.5 which explains the structure of the extension, followed by section 6.6 which provides detailed overview of the source codes.

Another important part for a user of the extension is to have a complete example of a deployment. This is provided in the section 6.8.

Last but not least, testing is important for every application and this extension is no exception. Testing options are offered by section 6.9.

Chapter 7

Conclusion

This extension can be a great simplification of writing clustered applications where sharing state of beans matters. Also, as CDI is quite young standard, this extension can be used as a good example of writing new extensions. The application is open-sourced and as such as long as there will be interest in using and developing the application further, this work will not die.

The biggest challenge at the moment will be to prepare high-quality test-suite and announce the extension somewhere in the JBoss community.

At last but not least this extension shows how new technologies, such as CDI or Infinispan make writing of such complex tasks as contextual data clustering easy.

Bibliography

[1] Gavin King. JSR-299: Contexts and Dependency Injection for the Java EE platform. Technical report, JSR-299 Expert Group, December 2009.

[2] Gavin King, Pete Muir, Dan Allen, and David Allen. *Weld - JSR-299 Reference Implementation*. Red Hat, January 2012.

[3] Gavin King, Pete Muir, Norman Richards, Shane Bryzak, Michael Yuan, Mike Youngstrom, Christian Bauer, Jay Balunas, Dan Allen, Max Rydahl Andersen, Emmanuel Bernard, Nicklas Karlsson, Daniel Roth, Matt Drees, Jacob Orshalick, Denis Forveille, Marek Novotny, , and Jozef Hartinger. *A Framework for Enterprise Java*. JBoss.org, September 2012.

[4] Red Hat. *Infinispan User Guide*.

Appendix A

Extension's code structure

```
cdi-cluster/src/main/
├── java
│   └── cz
│       └── vutbr
│           └── fit
│               └── bp
│                   └── xkovar57
│                       ├── ClusterContext.java
│                       ├── ClusterContextExtension.java
│                       ├── interceptors
│                       │   └── ClusteredOperationInterceptor.java
│                       ├── annotations
│                       │   ├── ClusterScoped.java
│                       │   └── ClusteredOperation.java
│                       ├── cache
│                       │   ├── CacheNotFoundException.java
│                       │   └── configuration
│                       │       ├── JndiCacheConfigurator.java
│                       │       └── CacheConfigurator.java
│                       ├── common
│                       │   └── CacheProvider.java
│                       └── serialization
│                           ├── XStreamObjectSerializer.java
│                           ├── InvalidTypeForSerializationException.java
│                           └── ObjectSerializer.java
└── resources
    └── META-INF
        ├── MANIFEST.MF
        ├── beans.xml
        └── services
            └── javax.enterprise.inject.spi.Extension
```

`ClusterContextExtension.java` registers the `@ClusterContext` annotation for further use by user applications

ClusterContext.java is the actual CDI context implementation where creating new beans and obtaining `@Inject`-able references is done

ClusteredOperationInterceptor.java is an interceptor which is invoked on methods annotated as `@ClusteredOperation`, subsection 6.7.2

ClusterScoped.java is the `@ClusterScoped` annotation

ClusteredOperation.java is the `@ClusteredOperation` annotation, subsection 6.8.5

CacheNotFoundException.java is an exception thrown in case the `Cache` not found

JndiCacheConfiguration.java is an implementation of `CacheConfigurator` interface which uses JNDI lookup to obtain a `Cache`

CacheConfigurator.java is an interface that must be implemented when a custom `Cache` provider needs to be implemented

CacheProvider.java is a class defining operation on the `Cache` which are used during invocations of `@ClusteredOperations`

XStreamObjectSerializer.java is an implementation of `ObjectSerializer` interface which uses XStream library to serialize/deserialize beans

InvalidTypeForSerializationException.java is an exception thrown in case of invalid input for serialization/deserialization methods

ObjectSerializer.java is an interface that must be implemented when a custom `Serializer` / `Deserializer` needs to be implemented, subsection 6.7.1

META-INF/MANIFEST.MF contains dependencies of JBoss AS logging and Infinispan modules

META-INF/beans.xml contains Alternatives for beans of which implementation depends on the user and deployment needs, subsection 6.7.3

META-INF/services/javax.enterprise.inject.spi.Extension contains the Extension class Fully-Qualified-Name(FQN)

Appendix B

Example application's code structure

```
cdi-cluster-test/src/main/
├── java
│   └── cz
│       └── vutbr
│           └── fit
│               └── bp
│                   └── xkovar57
│                       └── test
│                           ├── GetServlet.java
│                           ├── SetServlet.java
│                           ├── util
│                           │   └── Resources.java
│                           └── data
│                               └── ClusteredBean.java
├── resources
│   └── META-INF
│       ├── MANIFEST.MF
│       └── cdi-cache-jndi.properties
└── webapp
    ├── web.xml
    └── WEB-INF
        └── beans.xml
```